A Framework for Customisable Schema Evolution in Object-Oriented Databases

Awais Rashid
Computing Department, Lancaster University, Lancaster LA1 4YR, UK
awais@comp.lancs.ac.uk

Abstract

This paper describes an evolution framework supporting customisation of the schema evolution and instance adaptation approaches in an object database management system. The framework is implemented as an integral part of an interpreter for a language with a versioned type system and employs concepts from object-oriented frameworks and aspect-oriented programming to support flexible changes. Some example customisations currently implemented with the framework are also described.

1. Introduction

The schema of an object-oriented database (OODB) is subject to change over its lifetime. A number of schema evolution approaches have been proposed to accommodate such changes. These range from basic s*chema modification* (no versioning) [3, 9] to versioning of individual classes (*class versioning*) [14, 24] or whole schemas (*schema versioning*) [11, 16] through to mechanisms versioning partial, subjective views of the schema [1] or those based on superimposing one approach on another e.g. [18]. Similarly, a number of instance adaptation mechanisms – supporting simulated or physical conversion of objects upon schema evolution – have been proposed e.g. [9, 14, 22, 24]. Work has also been carried out to maintain behavioural consistency of applications as the schema evolves [4, 12, 13].

Traditionally, an object database management system (ODBMS) offers the maintainer one particular schema evolution approach coupled with a specific instance adaptation mechanism. For instance, CLOSQL [14] is a *class versioning* system employing *dynamic instance conversion* as the instance adaptation mechanism; ORION [3] employs *schema modification* and *transformation functions*; ENCORE [24] uses *class versioning* and *error handlers* to simulate instance conversion.

It has been argued that such "fixed" functionality does not serve "local" organisational needs effectively [21]. Organisations tend to have very specialised evolution requirements for their applications and these requirements can even vary across applications within an organisation. For one organisation (or application) it might be inefficient to keep track of change histories, hence making *schema modification* the ideal evolution approach. For another organisation (or application) maintenance of change histories and their granularity might be critical. Similarly, in one case it might be sufficient that instance conversion is simulated while in another scenario physical object conversion might be more desirable. The requirements can be specialised to the extent that custom variations of existing approaches might be needed.

This paper describes a framework to support such customisations. The work builds upon our previous work on supporting customisations in database systems [21], in particular providing flexible instance adaptation mechanisms [22]. The framework employs its own language, Vejal, with a versioned type system to support instance adaptation across schema changes. Aspect-oriented programming (AOP) techniques [7] and hot spots, as in object-oriented frameworks [8], are employed to support customisation of the schema evolution and instance adaptation approaches.

The next section provides a brief introduction to object-oriented frameworks and AOP. Section 3 describes the Vejal language underpining the framework. Sections 4 and 5 discuss customisation of the instance adaptation and schema evolution approach respectively. Section 6 concludes the paper and identifies directions for future work.

2. AOP and OO frameworks

A framework provides a basic system model for a particular application domain within which specialised applications can be developed. It consists of already coded pieces of software which are reused, the so called *frozen spots* and the flexible elements, the *hot spots*, which allow the user to adjust the framework to the needs of the concrete application [15]. Object-oriented frameworks [8] employ object-oriented techniques to support customisation and are categorised into *white-box* and *black-box* frameworks. The former employ inheritance to configure the hot spots hence requiring that the architecture of the framework is known to the application developers who build upon it. The latter hide the internal structure of the framework by employing *composition* as the customisation mechanism hence requiring knowledge of the hot spots only and not the internal structure of the framework. Black-box

frameworks, however, are harder to build than white-box frameworks. In practice there are few pure white or black box frameworks, and in most cases some hot spots are developed using a white-box approach while others use the black-box approach.

AOP [7] is a new programming paradigm which aims at separating concerns which cut across parts of a system. Examples of such crosscutting concerns include code handling synchronisation, persistence, debugging, security, resource sharing, distribution and memory management. Specific examples of crosscutting concerns in database systems include instance adaptation, change propagation and referential integrity semantics, and policies for version management, security, access control and cache management [19, 20, 22]. It is not possible to encapsulate such code within a single program module using conventional decomposition mechanisms – leading to tangled representations. For instance, in an object-oriented decomposition, code handling tracing, synchronisation or persistence is spread across several classes. With AOP crosscutting code is modularised using special constructs known as *aspects*. This promotes localisation of changes hence reducing development, maintenance and evolution costs. The links between aspects and classes are maintained by means of special reference points known as *join points*. An *aspect weaver* is used to compose the aspects and classes with respect to the join points. This composition may be carried out statically at compile time or dynamically at run time [10]. The join point specification and composition mechanism allow aspects to introduce additional behaviour into the program control flow at well-defined points. This makes AOP an ideal candidate to implement black-box customisations affecting a range of classes.

One of the leading AOP approaches is AspectJ [2], an aspect language for Java, developed at Xerox PARC and used in the implementation of the evolution framework described in this paper. In AspectJ, join points are nodes in a simple object call graph at run-time i.e. points at which an object receives a method call or has its fields referenced. Join points in AspectJ, therefore, include (among others) method/constructor calls and executions, field get and set, and exception handler execution. *Advices*, which are method-like constructs, are used to execute additional behaviour *before*, *after* or *around* a join point. Fig. 1 shows an example aspect in AspectJ (version 1.06) that displays tracing information (in this case the method signature) *before* and *after* executions of all the methods with any return type or arguments for objects of class `Main` and its subclasses (indicated by the "+" sign in the *pointcut* definition). Note that the class definitions, written in standard Java, do not require any special preparation e.g. hooks, for the aspect in fig. 1 to be employed. The aspect can be compiled in when tracing is needed and compiled out when it is not desirable.

```
public aspect Tracing {

    pointcut methodExecutions(): execution(* Main+.*(..));

    before(): methodExecutions() {

            System.out.println("Entering " + thisJoinPointStaticPart.getSignature());

    }

    after(): methodExecutions() {

            System.out.println("Leaving " + thisJoinPointStaticPart.getSignature());

    }

}
```

Fig. 1: A tracing aspect in AspectJ (1.06)

3. Framework infrastructure

The framework has been implemented as an integral part of an interpreter for Vejal. Applications are written in Vejal, whereas the framework and its concrete instantiations are implemented in AspectJ and Java.

Vejal has a two-level versioning identifier system (analogous to that used by the popular free source code control system CVS [6]). `C[1]` indicates class version 1 of class C while `C[s=1]` implies the class version of C that occurs in schema version 1. In Vejal, one version of a class may be present in multiple schema versions. In order to prevent unworkable schema versions being created, a new version of a class can only be present in all the future schema versions in which it is still compatible with the contracts of the other classes in the schema.

Vejal aims to support the programmer to specify arbitrarily complex transformations between class versions, in the form of *instance adaptation aspect*s [22]. These are essentially transparent view wrappers, written by the application programmer to present e.g. an instance of `Person[1]` as a `Person[2]`, which are invoked by the runtime environment automatically whenever an object needs to be adapted to a different class version. Crucially, they work by transforming data at the field level, and do not attempt to emulate methods (and nor do they require the application-specific evolution code to emulate methods) – the "real" methods are always used from the Vejal class version required by the application. Although this means an instance adaptation aspect breaks the encapsulation of the destination class version, this is arguably a good trade-off, because the alternative of emulating method behaviour leaves more room for error, and converting an object between class versions often requires knowledge of implementation details. There are no language restrictions on changes that can be made between one class version and the next. In particular, methods can be added, deleted and rewritten.

Vejal instance adaptation aspects are intended to support either simulated or physical conversions with

exactly the same aspect. Thus, the real adaptation approach in use is abstracted out (cf. section 4). If the system is configured to use simulated conversions for a particular class, the instance adaptation aspect is used to project a compatible view. On the other hand, if physical conversion is required, the runtime environment "scans through" the aspect to physically convert the object to the new class. In either case, *hidden fields* are used, if required, to store data from previous schemas that is invisible now but may become visible upon another adaptation [14]. Thus no data is lost due to destructive conversions unless a previous schema is itself deleted.

4. Customisable instance adaptation

The framework supports plugging in different approaches to instance adaptation. As shown in fig. 2, the hot spots are exposed using the abstract class *Adapter* in a white-box fashion. In order to customise the instance adaptation approach subclasses of *Adapter* have to concretise two abstract methods:

— **public abstract** SysObj createObj(ClassRef classRef, Env env)
This method is overridden to create the new object in case of physical conversion or circumvent the creation in case of simulated conversion. *SysObj* is the object to be created upon adaptation; *classRef* refers to the class the object is to be an instance of; *env* is the current environment of the running thread in the Vejal interpreter.

— **public abstract** SysObj convert(SysObj old, ConversionE conv, Env env)
This method is called by the underlying Vejal converter (*conv*), which invokes the instance adaptation aspects. It encapsulates the main logic for the particular instance adaptation approach offered by the concrete subclass of *Adapter*.

The three customisations currently implemented using the framework include (cf. fig. 2):

— **View Adapter (non-materialised views):** recalculates the view for each read operation rather than storing it. An exception is the case of entirely new fields which have to be stored as hidden fields in the original object.
— **Converter Adapter (physical conversion):** converts the object in-place and physically changes its class.
— **Elastic Converter Adapter (elastic conversion):** a variation of the basic physical conversion approach hence implemented as an aspect inheriting from Converter Adapter (aspects in AspectJ can inherit from classes but not vice versa). The aspect does not override any functionality in its superclass. Therefore, the object is physically converted. However, an advice in the aspect traps the join point when the converted object is about to be written to disk at the end of the transaction (black-box customisation) and reverts it to its previous schema version.

The customisations are beneficial because non-materialised views might be more appropriate in situations requiring reduction in memory consumption, while physical or elastic conversion might be more suitable in others because of its caching behaviour.

Other customisations are possible and, at least for all the approaches we have considered, do not require any changes whatsoever to the instance adaptation aspects. This is because instance adaptation aspects are written in a field-view-oriented style i.e. specifying how to obtain any given field of the destination object. This means that they can be used as-is in non-materialised views, or in other approaches the list of fields in a class can be iterated through and all fields pulled through the instance adaptation aspect like a sieve. This is essentially a special case of separating core code from caching concerns.

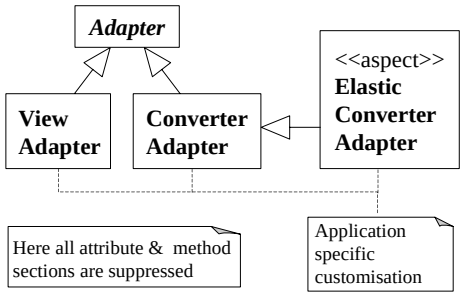

Fig. 2: UML diagram of classes and aspects for customisation of instance adaptation

5. Customisable evolution approach

Similar to customisation of instance adaptation, the framework exposes hot spots for customising the schema evolution approach in a white-box fashion (cf. fig. 3). However, instead of using a class hierarchy, an aspect hierarchy is employed. Advices in the aspects support the customisation by operating on other parts of the framework, mainly those pertaining to resolution of object references upon retrieval from the database (black-box customisation). The two abstract aspects, *Versioning Mode* and *Abstract N-Version Mode* expose a range of abstract methods that are concretised by the sub-aspects to customise the schema evolution approach.

Currently three versioning modes for schema evolution are supported (cf. fig. 3):

— **N-Version Mode:** unrestricted schema evolution is allowed. Instead of duplicating every single class on disk each time a schema is updated, only the classes that have changed are allocated new version numbers. This means that references to classes have to be indirect (versionless) on disk, because `Person` could refer to `Person[1]` or `Person[2]` depending on the schema version in use by an application.
— **One Version Mode:** only one schema version exists in the database. This provides some optimisations in case of applications where change histories are undesirable. All on-disk references to classes now refer directly to resolved classes, rather than indirect references as in N-version mode.
— **N-to-1 Transition Mode:** to go from N-version mode to one version mode. This is essential as a significant amount of disk activity is required to get the database into one version mode. Note that there is no transition period to go from one version mode to N-version mode. The transition is virtually instantaneous.

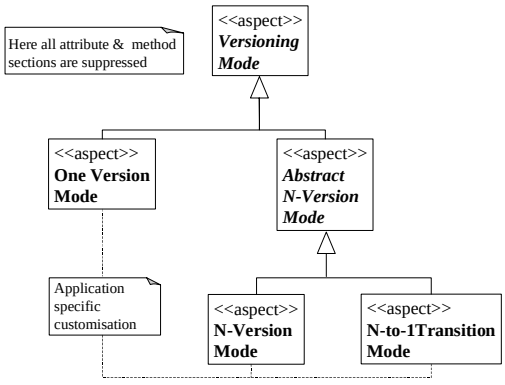

Fig. 3: UML diagram of aspects for customisation of schema evolution approach

6. Conclusion and future work

Existing schema evolution systems for object-oriented databases are inflexible resulting in organisations or applications having to adapt their evolution requirements to the functionality available. This leads to serious maintenance problems and evolution costs over the lifetime of the database with maintainers either opting for a cumbersome *dump and reload* approach or needing to develop an entirely new database to meet the evolution requirements [17]. The framework described in this paper offers the maintainers the ability to evolve the evolution mechanism offered by the ODBMS itself in step with evolution requirements. It supports customisation of the instance adaptation and schema evolution approaches. The two customisations can be carried out independently of each other or the instance adaptation aspects (which perform the actual adaptation in line with the approach specified using the framework). This flexibility has been achieved through a combination of AOP, hot spots, a language with a versioned type system and field-view-oriented instance adaptation aspects.

The use of AOP has not only made it possible to introduce additional behaviour into the system control flow in a modularised fashion but also allowed us to expose hot spots for "advanced" customisation. For instance, when customising the instance adaptation approach, most basic customisations (e.g. view adapter, converter adapter) can rely on the hot spots exposed in a white-box fashion (using inheritance). However, advanced customisations (e.g. elastic adapter) can manipulate other points through AOP.

The customisability of the framework to support other schema evolution and instance adaptation approaches and its ability to cope with unanticipated evolution requirements need to be validated in real-life applications and case studies. Our future work will, therefore, focus on this validation. Furthermore, the current implementation of the framework is based on the underlying ODBMS exposing an ODMG Java binding [5]. We aim to migrate to the Java Data Objects (JDO) standard [23] in order to support customisable evolution for a range of database systems including object-relational systems.

Acknowledgements

This work is supported by UK Engineering and Physical Sciences Research Council (EPSRC) grant GR/R08612. The author wishes to thank Robin Green for his contribution to the framework implementation.

References

[1] J. Andany, M. Leonard, and C. Palisser, "Management of Schema Evolution in Databases", 17th International Conference on Very Large Databases, 1991, Morgan Kaufmann, pp. 161-170.
[2] AspectJ Team, "AspectJ Project", http://www.eclipse.org/aspectj/, 2003.
[3] J. Banerjee, H.-T. Chou, J. F. Garza, W. Kim, D. Woelk, and N. Ballou, "Data Model Issues for Object-Oriented Applications", *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, pp. 3-26, 1987.
[4] P. L. Bergstein, "Maintenance of Object-Oriented Systems during Structural Evolution", *TAPOS - Theory and Practice of Object Systems*, Vol. 3, No. 3, pp. 185-212, 1997.
[5] R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russel, O. Schadow, T. Stenienda, and F. Velez, *The Object Data Standard: ODMG 3.0*: Morgan Kaufmann, 2000.

[6] "Concurrent Versions System", http://www.cvshome.org/, 2003.
[7] T. Elrad, R. Filman, and A. Bader (eds.), "Theme Section on Aspect-Oriented Programming", *Communications of ACM*, Vol. 44, No. 10, 2001.
[8] M. E. Fayad and D. C. Schmidt, "Object-Oriented Application Frameworks", *Communications of the ACM*, Vol. 40, No. 10, pp. 32-38, 1997.
[9] F. Ferrandina, T. Meyer, R. Zicari, and G. Ferran, "Schema and Database Evolution in the O2 Object Database System", 21st Conference on Very Large Databases, 1995, Morgan Kaufmann, pp. 170-181.
[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", ECOOP, 1997, Springer-Verlag, Lecture Notes in Computer Science, 1241, pp. 220-242.
[11] W. Kim and H.-T. Chou, "Versions of Schema for Object-Oriented Databases", 14th International Conference on Very Large Databases, 1988, Morgan Kaufmann, pp. 148-159.
[12] G. N. C. Kirby, "Persistent Programming with Strongly Typed Linguistic Reflection", 25th International Conference on System Sciences, 1992, pp. 820-831.
[13] L. Liu, R. Zicari, W. Huersch, and K. J. Lieberherr, "The Role of Polymorphic Reuse Mechanisms in Schema Evolution in an Object-Oriented Database", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 1, pp. 50-67, 1997.
[14] S. Monk and I. Sommerville, "Schema Evolution in OODBs Using Class Versioning", *ACM SIGMOD Record*, Vol. 22, No. 3, pp. 16-22, 1993.
[15] W. Pree, *Design Patterns for Object Oriented Software Development*: Addison Wesley, 1994.
[16] Y.-G. Ra and E. A. Rundensteiner, "A Transparent Schema-Evolution System Based on Object-Oriented View Technology", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 4, pp. 600-624, 1997.
[17] A. Rashid, "A Database Evolution Approach for Object-Oriented Databases", *PhD Thesis*, Computing Department, Lancaster University, UK, 2000.
[18] A. Rashid, "A Database Evolution Approach for Object-Oriented Databases", IEEE International Conference on Software Maintenance (ICSM), 2001, IEEE Computer Society Press, pp. 561-564.
[19] A. Rashid, "A Hybrid Approach to Separation of Concerns: The Story of SADES", 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001, Springer-Verlag, Lecture Notes in Computer Science, 2192, pp. 231-249.
[20] A. Rashid and E. Pulvermueller, "From Object-Oriented to Aspect-Oriented Databases", 11th International Conference on Database and Expert Systems Applications (DEXA), 2000, Springer-Verlag, Lecture Notes in Computer Science, 1873, pp. 125-134.
[21] A. Rashid and P. Sawyer, "Aspect-Orientation and Database Systems: An Effective Customisation Approach", *IEE Proceedings - Software*, Vol. 148, No. 5, pp. 156-164, 2001.
[22] A. Rashid, P. Sawyer, and E. Pulvermueller, "A Flexible Approach for Instance Adaptation during Class Versioning", ECOOP 2000 Symposium on Objects and Databases, 2000, Springer-Verlag, Lecture Notes in Computer Science, 1944, pp. 101-113.
[23] R. Roos, *Java Data Objects*: Addison Wesley, 2002.
[24] A. H. Skarra and S. B. Zdonik, "The Management of Changing Types in an Object-Oriented Database", 1st OOPSLA Conference, 1986, pp. 483-495.